

JavaScript Objects

(Detailed notes yet beginner friendly)

Why Objects exist in JavaScript or in General ?

Start from the most basic problem

Imagine you are programming without objects.

You only have:

- numbers
- strings
- maybe arrays

Now suppose you want to represent a user:

Name: Saint Kabir

Age: 59

Email: saintkabir@email.com

Without objects, what can you do?

Option 1: Separate variables

```
let name = "Saint Kabir";
```

```
let age = 59;
```

```
let email = "saintkabir@email.com";
```

Problem:

- These variables are not grouped
- Easy to mess up
- Hard to pass around

Option 2: Use arrays

```
let user = ["Saint Kabir", 59, "saintkabiremail.com"];
```

Problems:

- What is user[1]? Age? Email?
- You must remember positions
- Not readable
- Not scalable

Core Problem Identified

We need a way to:

- Group related data
- Give meaning to each piece
- Access data without guessing

Step 2: The Idea That Led to Objects

Instead of:

```
["Saint Kabir", 59, "email"]
```

We want:

```
"name" → "Saint Kabir"
```

```
"age" → 59
```

```
"email" → "..."
```

This leads birth of objects

What is a an Object ?

- An object is a collection of key-value pairs where keys are strings or Symbols and values can be any type
- It is in key-values pairs.

Think of a real-world object, like a person. A person has properties with labels and values:

- Name: "Alice"
- Age: 30
- Is Student: true

A JavaScript object is the exact same thing: a container for named values, which we call key-value pairs.

- The key is the label (Keys are always strings or Symbols internally).
- The value is the data (it can be any data type: a number, a string, a boolean, an array, or even another object).

```
// This is an object representing a person

let person =
  { name: "Alice", // Key is "name", Value is "Alice"
    age: 30, // Key is "age", Value is 30
    isStudent: true // Key is "isStudent", Value is true
  };
```

A. Creating a Object

The most common and best way to create an object is using the object literal syntax: curly braces {}

```
let username = "Rajessfh";

// This is an object representing a person
let person =

  { name: "Alice", // Key is "name", Value is "Alice"
    age: 30, // Key is "age", Value is 30
    isStudent: true // Key is "isStudent", Value is true

  };
```

B. Accessing (Reading) Properties

There are two ways to access objects

1. **Dot Notation (.)** : The simplest and most common way. It's clean and easy to read.

```
console.log(user.username); // "js_dev"
```

Limitation : Dot notation only works for keys that are valid JavaScript identifiers (no spaces, hyphens, or starting with a number).

2. **Bracket Notation** : The more powerful and flexible way. The key inside the brackets is a string.

```
console.log(user["loginCount"]); // 57
```

// You MUST use bracket notation for keys with special characters.

```
console.log(user["is-premium-member"]); // true
```

CRITICAL USE CASE: Bracket notation allows you to use a variable to determine which property to access.

```
let propertyToAccess = "username";
```

```
console.log(user[propertyToAccess]); // "js_dev" (This is impossible with dot notation)
```

C. Updating and Adding (Creating) Properties

You use the same syntax for both. If the property exists, it's updated. If it doesn't, it's created.

```
const book = {  
  title: "The Hobbit"  
};  
  
// Update an existing property  
book.title = "The Lord of the Rings";  
  
// Add a new property  
book.author = "J.R.R. Tolkien";  
book.pages = 1178;  
  
console.log(book); // { title: "The Lord of the Rings",  
author: "J.R.R. Tolkien", pages: 1178 }
```

D. Deleting Objects

Use the **delete** keyword.

```
delete book.pages;  
  
console.log(book); // { title: "The Lord of the Rings",  
author: "J.R.R. Tolkien" }
```

When a function is a value inside an object, it's called a method. Methods give objects behavior.

The **this** Keyword

First Thought: this is a shortcut that means "this object right here."

Inside a method, the **this** depends on how a function is called, not where it is defined..

```
const user = {  
  name: "Alice",  
  greeting: function() {  
    // Here, `this` refers to the 'user' object.  
    console.log(`Hello, my name is ${this.name}.`);  
  }  
};  
  
user.greeting(); // Outputs: "Hello, my name is Alice."
```

When we call **user.greeting()**, this becomes **user**. This allows methods to be dynamic and access the data of their own object.

How do you get all the keys and values from an object?

A. The **for...in** Loop (The Old Way)

This loop iterates over the keys (property names) of an object.

```
const car = {  
  make: "Honda",  
  model: "Civic",  
  year: 2021  
};  
  
for (const key in car) {  
  console.log(`Key: ${key}, Value: ${car[key]}`);  
}
```

Another Problem: **for...in** also iterates over properties from the object's prototype chain, which can be unexpected. For this reason, the modern methods below are usually preferred.

B. The Modern Object Methods (The Better Way)

These methods return arrays, which you can then loop over with a **for...of** loop.

- **Object.keys(obj)** : Returns an array of the object's own property keys.
- **Object.values(obj)** : Returns an array of the object's own property values.
- **Object.entries(obj)** : Returns an array of [key, value] pairs.

```
const car = {
  make: "Honda",
  model: "Civic",
  year: 2021
};

console.log(Object.keys(car)); // ["make", "model",
"year"]
console.log(Object.values(car)); // ["Honda", "Civic",
2021]

// The .entries() method is perfect with a for...of loop
for (const [key, value] of Object.entries(car)) {
  console.log(`${key}: ${value}`);
}
```

The Modern Way (Recommended): Object.keys, values, entries

First Thought : "I can't loop over an object directly, so I'll ask for a list of its keys (or values) and loop over that list instead."

This is the core idea. These Object methods convert the parts of your object into an **array**, and we already know how to loop over arrays (**using for...of**) .

1. `Object.keys(obj)` - Looping Over Just the Keys

This method gives you an array of the object's keys (the property names).

```
const keys = Object.keys(user);
// keys is now: ["name", "age", "isStudent"]

// Now we can use a simple for...of loop on this new array
for (const key of keys) {
  // To get the value, we use bracket notation on the original
  // object
  const value = user[key];

  console.log(`The key is "${key}" and the value is "${value}".`);
}
```

Output

```
The key is "name" and the value is "Alice".
The key is "age" and the value is "30".
The key is "isStudent" and the value is "true".
```

2. **Object.values(obj)**- Looping Over Just the Values

This method gives you an array of the object's values. This is useful if you don't care about the property names.

```
const values = Object.values(user);  
// values is now: ["Alice", 30, true]  
  
for (const value of values) {  
  console.log(`The value is: ${value}`);  
}
```

Output

```
The value is: Alice  
The value is: 30  
The value is: true
```

3. **Object.entries(obj)** - The Best of Both Worlds

This is often the most useful method. It gives you an array of `[key, value]` pairs. Each item in the array is a smaller array containing the key at index 0 and the value at index 1.

```
const entries = Object.entries(user);  
//   entries is now:  
//   [  
//     ["name", "Alice"],  
//     ["age", 30],  
//     ["isStudent", true]  
//   ]
```

This works perfectly with a **for...of** loop and a technique called **destructuring**, which lets you unpack the key and value into their own variables right in the loop definition.

```
// This is the cleanest and most common modern pattern  
for (const [key, value] of Object.entries(user)) {  
  console.log(`Key: ${key}, Value: ${value}`);  
}
```

Output

```
Key: name, Value: Alice  
Key: age, Value: 30  
Key: isStudent, Value: true
```

The Traditional Way: for...in Loop

First Thought : "For each property key in this object, do this."

The for...in loop iterates directly over the property names (keys) of an object.

Syntax

```
const user = {  
  name: "Alice",  
  age: 30,  
  isStudent: true  
};  
  
for (const key in user) {  
  // 'key' will be "name", then "age", then "isStudent"  
  const value = user[key];  
  console.log(`Key: ${key}, Value: ${value}`);  
}
```

Why for...in is Usually Avoided

The for...in loop has a major "gotcha" that makes it less safe than the Object methods. It will also loop over properties that the object inherits from its prototype

This is an advanced concept, but a simple example can show the problem.

```
// Let's modify the master Object prototype (NEVER DO THIS IN  
REAL CODE!)  
Object.prototype.isInherited = true;  
  
const person = { name: "Bob" };  
  
console.log("--- Using for...in ---");  
for (const key in person) {  
  console.log(key); // Outputs: "name", AND "isInherited"  
}  
  
console.log("\n--- Using Object.keys ---");  
console.log(Object.keys(person)); // Outputs: ["name"] (It only  
sees its OWN properties)
```

Because **for...in** sees the inherited `isInherited` property, it can cause unexpected behavior. The **Object.keys()** method correctly ignores it. To make **for...in** safe, you have to add an extra check, which makes it clumsy.

Summary: Which Loop to Use?

Method	Best For...	Why?
<code>for (const [key, value] of Object.entries(obj))</code>	The default choice. Getting both the key and the value.	Clean, readable, uses modern syntax, and is safe (only iterates own properties).
<code>for (const key of Object.keys(obj))</code>	Getting only the keys, and then looking up the values.	Safe and clear. A good alternative to <code>entries</code> .
<code>for (const value of Object.values(obj))</code>	Getting only the values.	Simple and direct if you don't need the keys.
<code>for...in</code>	Generally avoid. Use only if you specifically <i>need</i> to inspect inherited properties.	Can cause bugs by iterating over the prototype chain. The other methods are safer.

A. Objects are a Reference Type

This is the same critical concept we saw with arrays. When you assign an object to another variable, you only copy the reference (the memory address). Both variables point to the same object.

```
let obj1 = { value: 10 };  
let obj2 = obj1; // Copies the reference, NOT the object.  
  
obj2.value = 20; // Mutates the single object.  
  
console.log(obj1.value); // 20 (The original was changed!)
```

B. How to Copy an Object (Shallow Copy)

To create a true copy of an object's top-level properties, use the **spread syntax** (`...`) or **Object.assign()**.

```
const original = { name: "Alice", age: 30 };  
  
// Using spread syntax (most common and modern)  
const copy = { ...original };  
  
copy.age = 31;  
  
console.log(original.age); // 30 (The original is safe!)  
console.log(copy.age);    // 31
```

(Note: This is a "shallow" copy. If the object contains other objects, those nested objects will still be references, not copies.)

C. Modern ES6+ Syntax (Syntactic Sugar)

- **Property Value Shorthand** : If you have a variable with the same name as a property key, you can just use the variable name once.

```
const name = "Alice";
const age = 30;

// Old way:
const userOld = { name: name, age: age };

// New way:
const userNew = { name, age }; // Much cleaner
```

- **Method Shorthand** : You can omit the **function** keyword when defining methods.

```
// Old way:
const userOld = {
  sayHi: function() { console.log("Hi"); }
};

// New way:
const userNew = {
  sayHi() { console.log("Hi"); }
};
```

- **Computed Property Names** : Use a variable as a property key during creation by putting it in brackets.

```
let propertyName = "email";
const user = {
  name: "Alice",
  [propertyName]: "alice@example.com"
};
console.log(user); // { name: "Alice", email: "alice@example.com" }
```

1. How to Copy an Object (Shallow vs. Deep)

This is a critical concept because objects are reference types. A simple assignment (let b = a) does not create a copy; it just makes a second variable point to the same object.

First Thought : "I don't want to change my original object. I need a brand new, separate copy to work with."

There are two kinds of copies: shallow and deep.

A. Shallow Copy (The 99% Use Case)

A shallow copy creates a new top-level object, but if any of the properties of the original object are themselves objects, it only copies the references to those nested objects.

```
const originalUser = {
  name: "Alice",
  age: 30,
  address: { // A nested object
    city: "New York"
  }
};

// Use the spread syntax to create a new object
const shallowCopy = { ...originalUser };

// --- Let's test the copy ---

// 1. Change a top-level primitive property in the copy
shallowCopy.age = 31;

console.log(originalUser.age); // 30 (The original is safe!)
console.log(shallowCopy.age); // 31 (The copy was changed)

// 2. Now, change a property in the NESTED object
shallowCopy.address.city = "London";

// The "gotcha" of a shallow copy:
console.log(originalUser.address.city); // "London" (The original was
also changed!)
```

Why did the original change? Because both **originalUser** and **shallowCopy** contain a reference to the exact same **address** object.

The Old Way : **Object.assign()**

This does the same thing as the spread syntax.

```
const shallowCopyAssign = Object.assign({}, originalUser);
```

B. Deep Copy (For Complex, Nested Objects)

First Thought : "Finally, a proper 'Clone' button for my objects."

The **structuredClone()** global function was introduced to solve this exact problem. It is designed to create a deep copy of a given value using the "structured clone algorithm," which is the same powerful algorithm that browsers use internally to copy data for features like **postMessage** (sending data between windows).

How it works : It recursively traverses the entire object, creating brand new copies of every object and array it finds, resulting in a completely independent clone.

```
const originalUser = {
  name: "Alice",
  age: 30,
  joined: new Date("2023-01-15"), // A Date object
  address: { // A nested object
    city: "New York"
  },
  roles: ["editor", "viewer"] // An array
};

// Use the built-in function to create a deep copy
const deepClone = structuredClone(originalUser);

// --- Let's prove it's a deep copy ---

// 1. Change a top-level primitive in the clone
deepClone.age = 31;

// 2. Change a property in the NESTED object
deepClone.address.city = "London";

// 3. Mutate the nested ARRAY
deepClone.roles.push("admin");

// 4. Mutate the nested DATE object
deepClone.joined.setFullYear(2024);

// --- Check the original object ---
// Everything is completely untouched!
console.log(originalUser.age);           // 30
console.log(originalUser.address.city);  // "New York"
console.log(originalUser.roles);         // ["editor", "viewer"]
console.log(originalUser.joined.getFullYear()); // 2023
```

Conclusion : structuredClone()

is the modern, correct, and recommended way to create a deep copy in JavaScript.

Browser and Node.js Support for structuredClone()

This is a relatively new feature, but it now has excellent support in all modern environments:

- Chrome: Version 98+
- Firefox: Version 94+
- Safari: Version 15.4+
- Node.js: Version 17.0.0+

You can safely use it today in any up-to-date browser or Node.js environment.

Limitations of **structuredClone()**

While it is extremely powerful, the structured clone algorithm is not a magical "clone everything" button. It is designed to clone data, not code or system resources.

It CANNOT clone:

- **Functions** : It will throw a `DataCloneError`. Functions have a "scope" and are not considered simple data.
- DOM Nodes: It will throw an error. You can't clone a piece of a webpage this way.
- **Class Instances (Prototypes)** : It will discard the object's prototype chain. The clone will be a plain object with all the same data properties, but it will no longer be an "instance of" your custom class.
- **Error objects.**

Example of a Failure:

```
const original = {
  name: "Alice",
  sayHi: function() { console.log("Hi"); }
};

try {
  const clone = structuredClone(original);
} catch (error) {
  console.error(error.name); // "DataCloneError"
  console.error(error.message); // "()" => { console.log("Hi"); } could
  not be cloned."
}
```

First Thought : "Instead of getting the whole box, just give me the specific items I want out of it."

Destructuring is a powerful syntax for "unpacking" values from arrays or properties from objects into distinct variables. It makes your code much cleaner and more readable.

A. Object Destructuring

This lets you pull properties out of an object by name.

```
const user = {
  id: 123,
  name: "Alice",
  email: "alice@example.com",
  profile: {
    isAdmin: true
  }
};

// --- Basic Destructuring ---
// Old way:
// const name = user.name;
// const age = user.age;

// New way with destructuring:
const { name, age } = user; // Create variables 'name' and 'age' from user
object.
console.log(name); // "Alice"
console.log(age); // undefined (because 'age' doesn't exist on the object)

// --- Renaming Variables ---
// What if you already have a variable named 'name'? You can rename it.
const { name: userName, email } = user;
console.log(userName); // "Alice"

// --- Setting Default Values ---
// What if a property might not exist? You can provide a default.
const { name: personName, role = "User" } = user;
console.log(personName); // "Alice"
console.log(role); // "User" (because 'role' didn't exist on the object)

// --- Nested Destructuring ---
// You can even pull from nested objects.
const { profile: { isAdmin } } = user;
console.log(isAdmin); // true
```

B. Array Destructuring

This lets you unpack values from an array by their position.

```
const scores = [98, 85, 100, 92];

// Get the first two scores
const [firstScore, secondScore] = scores;
console.log(firstScore); // 98
console.log(secondScore); // 85

// Skip items with a comma
const [winner, , thirdPlace] = scores;
console.log(winner); // 98
console.log(thirdPlace); // 100
```

Difference between Arrays & objects

JS

Object	Array
key-value	index-based
unordered	ordered
real-world entities	lists

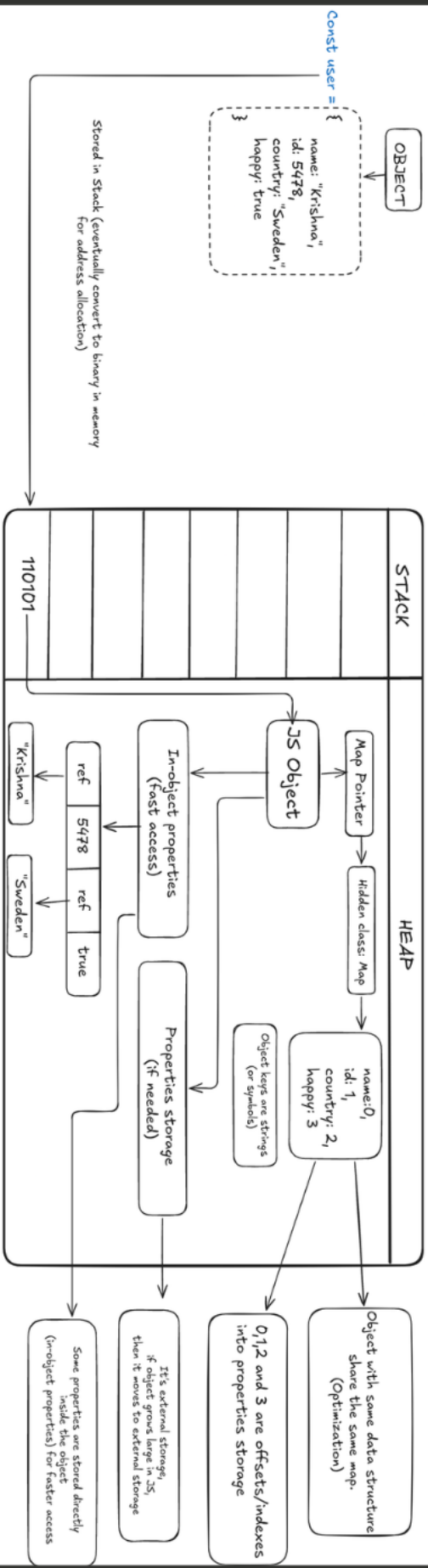
NOTE : Objects are NOT ordered

Objects are unordered collections (although modern JS maintains insertion order for most cases, you should not rely on it)."

How Objects stored in memory in JS ?

How Objects are Stored in JavaScript

Based on the V8 Engine architecture (Chrome/Node.js)



NOTE: JavaScript uses Dictionary mode, when object becomes dynamic, Ex- delete username
user.random = 20

Engine may switch to dictionary mode (hash table internally) when object becomes highly dynamic

Map pointer lets JavaScript skip property lookup and jump directly to the value using a pre-defined structure (hidden class).

CONCLUSION Object is stored in heap, but how it will be stored totally dependent on the nature of object.
JavaScript does NOT follow linearity, it's dynamic and dependent on object